Parallelizing Deadlock Resolution in Symbolic Synthesis of Distributed Programs *

Borzoo Bonakdarpour Fuad Abujarad Sandeep S. Kulkarni
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
Email: {borzoo,abujarad,sandeep}@cse.msu.edu

Abstract

Previous work has shown that there are two major complexity barriers in the synthesis of fault-tolerant distributed programs, namely generation of *fault-span*, the set of states reachable in the presence of faults, and, resolving *deadlock states*, where the program has no outgoing transitions. Although symbolic techniques can improve the performance of synthesis algorithms by orders of magnitude, efficient heuristics are still needed to overcome the aforementioned obstacles. Thus, motivated by the idea of partitioning the transition relation of distributed programs across multiple threads, in this paper, we introduce an efficient parallel (shared memory) algorithm for resolving deadlock states in symbolic synthesis of distributed programs. In spite of notorious resistance of symbolic algorithms for parallelization, experimental results show that our parallel algorithm exhibits *superlinear* performance improvement.

Keywords: Program transformation, Program synthesis, Parallel algorithm, Multi-core, Distributed programs, Deadlock resolution, Fault-tolerance.

1 Introduction

Automatically deriving programs that are correct-by-construction has been one of the most ambitious goals in computer science for several decades. Such automatic construction of programs is especially useful in dependable mission/safety-critical systems where *correctness* plays a crucial role. One way to achieve this goal is to use *program synthesis* techniques. Program synthesis is especially beneficial in program maintenance where system requirements constantly evolve and, thus, programs need to be revised. In the context of distributed systems, program synthesis is desirable when an existing program is subject to uncontrollable *faults*. Indeed, since it may be virtually impossible to anticipate all faults that a distributed program may be subject to at design time, it is highly advantageous for designers of fault-tolerant systems to have access to synthesis methods that incrementally *add* fault-tolerance to a given distributed fault-intolerant program. Intuitively, by a *fault-tolerant* program, we mean a program that meets its safety and liveness requirements in both absence and presence of faults. And, the corresponding synthesis problem focuses on analyzing the existing fault-intolerant program to add/remove transitions/actions so that the revised program is fault-tolerant. Note that by its nature, such synthesis algorithms are offline because they focus on transforming one program into another.

One crucial problem in program synthesis is the time and space complexity. To manage these complexities, in our previous work [1, 2], we proposed a set of enumerative and symbolic (BDD-based) techniques for adding fault-tolerance to existing distributed fault-intolerant programs. In order to synthesize a fault-tolerant program, the algorithms in [1,2] repeat a sequence of steps such as (1) generation of fault-span (the set of states reachable by program and fault transitions), (2) identifying and removing unsafe transitions, (3) resolving deadlock states, and (4) reconstructing invariant predicate, until a fixedpoint is reached. We also showed that symbolic techniques [2] improve the performance of synthesis by several orders of magnitude, paving the path for synthesizing moderate-sized programs with state space of size 10^{30} and beyond. Based on the analysis of the experimental results from [2], we observed that depending upon the structure of the given distributed intolerant program, performance of synthesis suffers from two major complexity obstacles, namely *generation of fault-span* and *resolution of deadlock states*. Thus, more efficient techniques are still needed to overcome the aforementioned bottlenecks. In this paper, we focus on the second problem, i.e., resolution of deadlock states. Deadlock resolution is especially crucial in the context of dependable systems, as it guar-

*This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2008		**00-00-2008 to 00-00-2008**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Parallelizing Deadlock Resolution in Symbolic Synthesis of Distributed Programs	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Michigan State University,Department of Computer Science and Engineering,East Lansing,MI,48824**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Previous work has shown that there are two major complexity barriers in the synthesis of fault-tolerant distributed programs, namely generation of fault-span, the set of states reachable in the presence of faults, and, resolving deadlock states, where the program has no outgoing transitions. Although symbolic techniques can improve the performance of synthesis algorithms by orders of magnitude, efficient heuristics are still needed to overcome the aforementioned obstacles. Thus, motivated by the idea of partitioning the transition relation of distributed programs across multiple threads, in this paper, we introduce an efficient parallel (shared memory) algorithm for resolving deadlock states in symbolic synthesis of distributed programs. In spite of notorious resistance of symbolic algorithms for parallelization, experimental results show that our parallel algorithm exhibits superlinear performance improvement.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**12**	

antees that the synthesized fault-tolerant program meets its liveness requirements even in the presence of faults.

1.1 The Deadlock Resolution Problem

We now describe the issue of deadlock resolution using the *Byzantine agreement* (denoted BA) problem [3]. We omit other steps involved in synthesizing a fault-tolerant version of BA (e.g., fault-span generation, preserving safety, and reconstructing invariant predicate), as they are not in the scope of this paper. BA consists of a *general*, say g, and three (or more) *non-general* processes: j, k, and l. Each process of BA maintains a decision d; for the general, the decision can be either 0 or 1, and for the non-general processes, the decision can be 0, 1, or $\bot$, where the value $\bot$ denotes that the corresponding process has not yet received the decision from the general. Each non-general process also maintains a Boolean variable f that denotes whether that process has finalized its decision. For each process, a Boolean variable b shows whether or not the process is Byzantine. In the *fault-intolerant* version of this program, each non-general process copies the decision from the general and then finalizes (outputs) that decision, provided it is non-Byzantine. A fault transition can cause a process to become Byzantine, if no other process is initially Byzantine. Also, a fault can change the d and f values of a Byzantine process. Let the sequence $\langle x_1, x_2, x_3, x_4\rangle$ denote the set of states with respect to decision value of processes, i.e., $x_1 = d.g$, $x_2 = d.j$, $x_3 = d.k$, and $x_4 = d.l$. In this notation, an overlined (respectively, underlined) d-value shows that the corresponding process has finalized its decision (respectively, is Byzantine). Now consider the following scenarios:

- Starting from a state s_0 in $\langle 1, \bot, \bot, 1\rangle$, where the general and process l agree on decision 1 and processes j and k are undecided, the program may reach the following sequence of states due to occurrence of faults (denoted $\dashrightarrow$) and execution of program actions (denoted $\rightarrow$): $\langle 1, \bot, \bot, 1\rangle \dashrightarrow \langle \underline{1}, \bot, \bot, 1\rangle \dashrightarrow \langle \underline{0}, \bot, \bot, 1\rangle \rightarrow \langle \underline{0}, 0, \bot, 1\rangle \rightarrow \langle \underline{0}, 0, 0, 1\rangle$. Let s_1 be a state in $\langle \underline{0}, 0, 0, 1\rangle$, where the Byzantine general g and non-general processes j and k agree on decision 0, but process l has decided on 1. Now, consider the tasks for a synthesis algorithm in dealing with state s_1. Note that no process can determine whether other processes have finalized their decision due to the issue of distribution. Thus, the synthesis algorithm rules out transitions that originate from s_1 and j finalizes its decision, as it would violate safety (i.e., agreement). Likewise, it cannot allow k and l to finalize either. We call states such as s_1 a *deadlock state*, since the program cannot proceed its execution. A synthesis algorithm can resolve this deadlock state by simply adding a *recovery* transition that changes the decision of l to 0 which results in reaching a legitimate state without violating safety. After adding such transitions, in the next iteration of the synthesis algorithm, we can allow j and k to finalize their decision after concluding that $\langle 0, 0, 0, \overline{1}\rangle$ (i.e., where l is not Byzantine and has finalized) is not reached.

- Now, consider the scenario where s_0 reaches the following sequence of states: $\langle 1, \bot, \bot, 1\rangle \rightarrow \langle 1, \bot, \bot, \overline{1}\rangle \dashrightarrow \langle \underline{1}, \bot, \bot, \overline{1}\rangle \dashrightarrow \langle \underline{0}, \bot, \bot, \overline{1}\rangle \rightarrow \langle \underline{0}, 0, \bot, \overline{1}\rangle \rightarrow \langle \underline{0}, 0, 0, \overline{1}\rangle$. Let s_2 be a state in $\langle \underline{0}, 0, 0, \overline{1}\rangle$, where non-general processes j and k agree with the Byzantine general on decision 0, but process l has finalized its decision on 1. Obviously, s_2 is also a deadlock state. However, unlike s_1 in the previous scenario, since process l has finalized its decision, we cannot resolve s_2 by adding safe recovery. One approach to deal with such deadlock states is to simply eliminate them (i.e., making them unreachable). However, since we require that during elimination of a deadlock state, no new deadlock states must be created, a respective deadlock resolution algorithm involves many backtracking steps. In particular, in order to resolve s_2, the algorithm needs to explore the reachability graph and remove the transition that allows a process to finalize its decision while there exist two undecided processes.

In [2], we observed that in order to automatically synthesize a fault-tolerant version of BA identical to the one by Lamport, Shostak, and Pease [3], 92% of the total synthesis time is spent to resolve deadlock states.

1.2 Contributions

With this motivation, in this paper, we introduce a parallel BDD-based algorithm for resolving deadlock states in distributed programs that are subject to a set of faults. We specifically design our algorithm for multiprocessor architectures with shared memory (e.g., multi-core processors) due to their availability in virtually any organization. Intuitively, our algorithm partitions the transition relation of the given intolerant program across multiple threads where each thread works on a different processor core. The algorithm makes no assumptions about the structure of a given program (e.g., set of transitions, number of distributed processes, or its reachable states) in order to resolve deadlock states. Thus, we expect the algorithm to be generally applicable to a wide variety of distributed programs. Our parallel algorithm tends to require more memory than its sequential version. However, based on our experimental results, unlike model checking, BDD-based synthesis algorithms *run out of time* before they *run out of memory*. Hence, the increased space

complexity is unlikely to be a bottleneck during synthesis.

We note that symbolic algorithms are known to be notoriously hard to parallelize due to the interdependence among data structures involved in such algorithms. As a matter of fact, while parallel implementations of symbolic model checkers are often successful in increasing available memory, the speedup gained from such techniques is limited. This is largely due to the irregular nature of the state-space generation task and the resulting high parallel overheads such as load imbalance and scheduling of small computations. Although some results in the literature (e.g., [4]) have concluded that parallelization of symbolic algorithms involves too many interrelated factors which leads to inefficiency in terms of speedups, we argue that parallelization based on partitioning the transition relation is remarkably efficient, as it can potentially minimize the interdependence among data structures such as BDDs. In fact, our experiments show that our parallel algorithm exhibits *superlinear speedup* as compared to the sequential algorithm.

Organization. The rest of the paper is organized as follows. In Sections 2 and 3, we present precise definitions for distributed programs, specifications, and fault-tolerance. We formally state the problem of synthesizing fault-tolerant programs in Section 4. Section 5 is dedicated to describe our parallel symbolic algorithm for deadlock resolution. Subsequently, experimental results and analysis are presented in Section 6. Related work is discussed in Section 7. Finally, we conclude in Section 8.

2 Distributed Programs and Specifications

Let $V = \{v_0, v_1 \cdots v_n\}$ be a finite set of Boolean variables. A *state* is determined by the function $s : V \mapsto \{true, false\}$, which maps each variable in V to either $true$ or $false$. Thus, we represent a state s by the conjunction $s = \bigwedge_{j=0}^{n} l(v_j)$ where $v_j \in V$ for all j, and $l(v_j)$ denotes a *literal*, which is either v_j itself or its negation $\neg v_j$. Since non-Boolean variables with finite domain D can be represented by $\log(|D|)$ Boolean variables, our notion of state is not restricted to Boolean variables.

Definition 2.1 (state predicate) A *state predicate* is a finite set of states. Formally, we specify a state predicate $S = \{s_0, s_1 \cdots s_m\}$ by the disjunction $S = \bigvee_{i=0}^{m}(s_i)$. ∎

Observe that although the formula defined in Definition 2.1 is in disjunctive normal form, one can represent a state predicate by any equivalent Boolean expression. We denote the membership of a state s in a state predicate S by $s \models S$.

A *transition* is a pair of states of the form (s, s') specified as a Boolean formula as follows. Let V' be the set $\{v' \mid v \in V\}$ (called *primed variables*). Primed variables are meant to show the new value of variables prescribed by a transition. Thus, we define a transition (s, s') by the conjunction $s \wedge s'$ where $s' = \bigwedge_{j=0}^{n} l(v'_j)$ such that $v'_j \in V'$ for all j.

Definition 2.2 (transition predicate) A *transition predicate* P is a finite set of transitions $\{(s_0, s'_0), (s_1, s'_1) \cdots (s_m, s'_m)\}$ formally defined by $P = \bigvee_{i=0}^{m}(s_i \wedge s'_i)$. We denote the membership of a transition (s, s') in a transition predicate P by $(s, s') \models P$. ∎

Notation. Let X be a state predicate. We use $\langle X \rangle'$ to denote the state predicate obtained by replacing all variables that participate in X by their corresponding primed variables. Also, let P be a transition predicate. We use $Guard(P)$ to denote the source state predicate of P (i.e., $s \models Guard(P)$ iff $\exists s' :: (s, s') \models P$). ∎

Definition 2.3 (closure) Let P be a transition predicate and S be a state predicate. We say that a state predicate S is *closed* in P iff $\bigwedge_{(s,s') \models P}((s \models S) \Rightarrow (s' \models \langle S \rangle'))$ holds. ∎

Definition 2.4 (process) A process j is specified by the tuple $\langle V_j, P_j, R_j, W_j \rangle$ where V_j is a set of variables, P_j is a transition predicate in the set of all possible states obtained from V_j (called *state space*), R_j is a set of variables that j can read, and W_j is a set of variables that j can write such that $W_j \subseteq R_j \subseteq V_j$ (i.e., we assume that j cannot blindly write a variable). ∎

Write restrictions. Let $\langle V_j, P_j, R_j, W_j \rangle$ be a process and $v(s)$ denote the value of a variable v in state s. Clearly, P_j must be disjoint from the following transition predicate: $NW_j = \bigvee_{(s,s')} \bigvee_{v \notin W_j}(v(s) \neq v(s'))$.

Read restrictions. Let $\langle V_j, P_j, R_j, W_j \rangle$ be a process, v be a variable in V_j, and $(s_0, s'_0) \models P_j$ where $s_0 \neq s'_0$. If v is not in R_j, then j must include a corresponding transition from all states s_1 where s_1 and s_0 differ only in the value of v. Let (s_1, s'_1) be one such transition. Now, it must be the case that s'_0 and s'_1 are identical except for the value of v. And, value of v must be the same in s_1 and s'_1. For instance, let $V_j = \{a, b\}$ and $R_j = \{a\}$. Thus, since j cannot read b, the transition $\neg a \wedge \neg b \wedge a' \wedge \neg b'$ and the transition $\neg a \wedge b \wedge a' \wedge b'$ have the same effect as far as j is concerned. Thus, each transition (s_0, s'_0) in P_j is associated with the following *group predicate*:

$$Group_j(s_0, s'_0) = \bigvee_{(s_1, s'_1)} (\bigwedge_{v \notin R_j}(v(s_0) = v(s'_0)) \wedge (v(s_1) = v(s'_1)) \wedge \bigwedge_{v \in R_j}(v(s_0) = v(s_1)) \wedge (v(s'_0) = v(s'_1)))$$

Definition 2.5 (program) A *program* P is specified by a set Pr of processes. We require that the state space of

all processes must be identical (i.e., $\forall i, j \in Pr :: V_i = V_j$). Thus, the state space of P is identical to the state space of its processes as well. For simplicity, we refer to a program P by the disjunction of its processes' transition predicates, i.e., $P = \bigvee_{j \in Pr}(P_j)$. ∎

To concisely write the transitions in a process, we use *guarded commands* (also called *actions*). A guarded command is of the form $L :: g \longrightarrow st$, where L is a label, g is a state predicate (called *guard*), and st is a *statement* that describes how the program state is updated. Thus, an action $g \longrightarrow st$ denotes the transition predicate $\{(s, s') \mid s \Rightarrow g$ and s' is obtained by changing s as prescribed by $st\}$.

Example (Byzantine agreement). Following the description of the Byzantine agreement program (denoted BA) in the introduction, BA consists of a general process g and three non-general processes j, k, and l. The state space of each process is obtained by variables in

$V = \{d.g, d.j, d.k, d.l\} \cup$ (decision variables)
$\{f.j, f.k, f.l\} \cup$ (finalized?)
$\{b.g, b.j, b.k, b.l\}$. (Byzantine?)

The transition predicate of a non-general process, say j, is specified by the following two actions:

$BA1_j :: (d.j = \bot) \wedge (f.j = \mathit{false}) \longrightarrow d.j := d.g$
$BA2_j :: (d.j \neq \bot) \wedge (f.j = \mathit{false}) \longrightarrow f.j := \mathit{true}$

Since the general process only provides a decision, its transition predicate is empty. The sets of variables that a non-general processes, say j, is allowed to read and write are $R_j = \{b.j, d.j, f.j, d.k, d.l, d.g\}$ and $W_j = \{d.j, f.j\}$, respectively.

Definition 2.6 (computation) A sequence of states, $c = \langle s_0, s_1 \cdots \rangle$, is a *computation* of program P iff the following two conditions are satisfied: (1) $\forall i \geq 0 : (s_i, s_{i+1}) \models P$, and (2) if c is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \models P$. ∎

We distinguish between a terminating computation and a *deadlocked* computation. Precisely, when a computation c *terminates* in state s_l, we include the transition (s_l, s'_l) in P, i.e., c can be extended to an infinite computation by stuttering at s_l. On the other hand, if there exists a state s_d such that there is no outgoing transition (or a self-loop) from s_d then s_d is a *deadlock* state.

Definition 2.7 (deadlock state) We say that a state s in program P is a *deadlock state* iff for all states s' in the state space of P, $(s, s') \not\models P$. ∎

2.1 Specification and Invariant

A *specification* $SPEC$ is a set of infinite sequences of states. We now define what it means for a program to satisfy a specification. We note that throughout the paper, we assume that state space of a program and its specification are identical.

Definition 2.8 (satisfies) Let P be a program, S be a state predicate, and $SPEC$ be a specification. We say that P *satisfies* $SPEC$ *from* S iff (1) S is closed in P, and (2) for all computations $c = \langle s_0, s_1 \cdots \rangle$ of P, where $s_0 \models S$, c is in $SPEC$. ∎

Definition 2.9 (invariant) Let P be a program, $SPEC$ be a specification, and S be a state predicate where $S \neq \mathit{false}$. We say that S is an *invariant predicate of* P *for* $SPEC$ iff P satisfies $SPEC$ from S. ∎

Observe that the notion of *satisfies* characterizes the property of infinite sequences with respect to a program. In order to characterize finite sequences, we introduce the notion of *maintains*.

Definition 2.10 (maintains) Let $SPEC$ be a specification, P be a program, and S be a state predicate. We say that program P *maintains* $SPEC$ from S iff (1) S is closed in P, and (2) for all computation prefixes α of P that starts from S, there exists a sequence of states β such that $\alpha\beta$ is in $SPEC$. Otherwise, we say that P *violates* $SPEC$. ∎

We let the specification consist of a *safety specification* and a *liveness specification*. Following Alpern and Schneider [5], safety specification can be characterized by a set of *bad prefixes* that should not occur in any computation. Throughout this paper, we let the length of such bad prefixes be two, i.e., a set of *bad transitions* denoted by transition predicate $SPEC_{bt}$. Thus, the safety specification can be formally defined by the set $SPEC_{\overline{bt}}$ of infinite sequences, such that no infinite sequence contains a transition in $SPEC_{bt}$.

A liveness specification of $SPEC$ is a set of infinite sequences of states that meets the following condition: for each finite sequence of states α there exists a suffix β such that $\alpha\beta \in SPEC$. In our synthesis problem (cf. Section 4), we begin with an initial program that satisfies its specification (including the liveness specification). As mentioned earlier, the focus of this paper is on developing a parallel algorithm that resolves reachable deadlock states of a program in the presence of faults. Clearly, such deadlock resolution is crucial in order to ensure that any finite computation of the synthesized program can be extended to an infinite computation that is in $SPEC$. In other words, our synthesis method preserves the liveness specification. Hence, the liveness specification need not be specified explicitly.

Notation. Whenever the specification is clear from the context, we will omit it; thus, "S is an invariant of P" abbreviates "S is an invariant predicate of P for $SPEC$". ∎

Example (cont'd). The safety specification of BA requires *validity* and *agreement*. *Validity* requires that if the general is non-Byzantine then the final decision of a non-Byzantine process must be the same as that of the

general. And, *agreement* requires that the final decision of any two non-Byzantine processes must be equal. Finally, once a non-Byzantine process finalizes (outputs) its decision, it cannot change it. Thus, the following transition predicate forms the safety specification, where p and q range over non-general processes:

$$
\begin{array}{l}
SPEC_{bt_{BA}} = \\
\quad (\exists p :: \neg b'.g \wedge \neg b'.p \wedge (d'.p \neq \bot) \wedge f'.p \wedge (d'.p \neq d'.g)) \ \vee \\
\quad (\exists p, q :: \neg b'.p \wedge \neg b'.q \wedge f'.p \wedge f'.q \wedge (d'.p \neq \bot) \wedge \\
\qquad\qquad (d'.q \neq \bot) \wedge (d'.p \neq d'.q)) \ \vee \\
\quad (\exists p :: \neg b.p \wedge \neg b'.p \wedge f.p \wedge ((d.p \neq d'.p) \vee (f.p \neq f'.p)))
\end{array}
$$

The invariant predicate of the Byzantine agreement program consists of the following states. First, we consider the set of states where the general is non-Byzantine. In this case, one of the non-general processes may be Byzantine. However, if a non-general process, say j, is non-Byzantine, it is necessary that $d.j$ be initialized to either $\bot$ or $d.g$. Also, a non-Byzantine process cannot finalize its decision if its decision equals $\bot$. Moreover, we consider the set of states where the general is Byzantine. In this case, g can change $d.g$ value arbitrarily. It follows that if other processes are non-Byzantine and $d.j$, $d.k$ and $d.l$ are initialized to the same value that is different from $\bot$, the program satisfies $SPEC_{bt_{BA}}$. Thus, the invariant predicate is as follows:

$$
\begin{array}{l}
S_{BA} = \\
\quad \neg b.g \wedge (\neg b.j \vee \neg b.k) \wedge (\neg b.k \vee \neg b.l) \wedge (\neg b.l \vee \neg b.j) \ \ \wedge \\
\quad (\forall p :: \neg b.p \Rightarrow (d.p = \bot \vee d.p = d.g)) \ \wedge \\
\quad (\forall p :: (\neg b.p \wedge f.p) \Rightarrow (d.p \neq \bot)) \ \vee \\
\quad b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \wedge (d.j = d.k = d.l \ \ \wedge d.j \neq \bot)
\end{array}
$$

An alert reader can easily verify that BA satisfies $SPEC_{bt_{BA}}$ from S_{BA}.

3 Fault Model and Fault-Tolerance

Following Arora and Gouda [6], the faults that a program P is subject to are systematically represented by a transition predicate F in the state space of P.

Example (cont'd). The fault transitions that affect a process, say j, of BA are as follows: (We include similar actions for k, l, and g)

$$
\begin{array}{lll}
F1 :: \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l & \longrightarrow & b.j := true \\
F1 :: b.j & \longrightarrow & d.j, f.j := 0|1, false|true
\end{array}
$$

where $d.j := 0|1$ means that $d.j$ could be assigned either 0 or 1. In case of the general process, the second action does not change the value of any f-variable.

Definition 3.1 (fault-span) Given a program P, faults F, and invariant S, we say that a state predicate T is an F-span (read as *fault-span*) of P from S iff the following two conditions are satisfied: (1) $S \Rightarrow T$, and (2) T is closed in $P \vee F$. ∎

Just as we defined the computation of P, we say that a sequence of states, $\langle s_0, s_1 \cdots \rangle$, is a *computation of P in the presence of F* iff the following three conditions are satisfied: (1) $\forall j > 0 :: (s_{j-1}, s_j) \models (P \vee F)$, (2) if $\langle s_0, s_1 \cdots \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \models P$, and (3) $\exists n \geq 0 :: (\forall j > n :: (s_{j-1}, s_j) \models P)$.

Definition 3.2 (fault-tolerance) Let P be a program with invariant S, F be a set of faults, and $SPEC$ be a specification. We say that P is *F-tolerant* (read as *fault-tolerant*) *to $SPEC$ from S* iff the following two conditions hold: (1) P satisfies $SPEC$ from S, and (2) there exists T such that (i) T is an F-span of P from S, (ii) $P \vee F$ maintains $SPEC$ from T, and (iii) every computation of $P \vee F$ that starts from a state in T has a state in S. ∎

4 The Synthesis Problem

Given are a program P with invariant S, a class of faults F, and specification $SPEC$ such that P satisfies $SPEC$ from S. Our goal is to find a program P' with invariant S' such that P' is F-tolerant to $SPEC$ from S'. In order to capture the requirement that our synthesis method only adds fault-tolerance and does not add new behaviors *in the absence of faults*, we introduce the notion of projection.

Definition 4.1 (projection) The *projection* of program P on state predicate S, denoted as $P|S$, is the program (i.e., transition predicate) $\bigvee_{(s,s') \models P}((s \models S) \wedge (s' \models \langle S \rangle'))$. I.e., $P|S$ consists of transitions of P that start in S and end in S. ∎

Now, observe that:

1. If S' contains states that are not in S then, in the absence of faults, P' may include computations that start outside S. Since we require that P' satisfies $SPEC$ from S', it implies that P' is using a new way to satisfy $SPEC$ in the absence of faults. Thus, we require that $S' \Rightarrow S$.
2. If $P'|S'$ contains a transition that is not in $P|S'$ then P' can use this transition in order to satisfy $SPEC$ in the absence of faults. Thus, we require that $(P'|S') \Rightarrow (P|S')$.

Following the above observations, the synthesis problem is as follows.

Problem statement. Given P, S, F, and $SPEC$ such that P satisfies $SPEC$ from S. Identify P' and S' such that:

$(C1)$ $S' \Rightarrow S$,
$(C2)$ $(P'|S') \Rightarrow (P|S')$, and
$(C3)$ P' is F-tolerant to $SPEC$ from S'. ∎

Notice that the third condition of the synthesis problem implies that every computation of P' that starts from a state in the fault-span of P', say T', has to be infinite (cf. Definition 3.2). Hence, T' cannot include any deadlock states. In the next section, we introduce our parallel algorithm for resolving deadlock states reachable from S using transitions in $P \vee F$. This algorithm can be used as a building block of algorithms for synthesizing P' and S'.

5 Parallel Symbolic Resolution of Deadlock States

In this section, we present our parallel BDD-based algorithm for resolving deadlock states reachable in the presence of faults in a distributed program. A major barrier in such parallelization is that BDD manipulation packages are not reentrant due to data structures shared across several BDDs (e.g., a hash table that stores all BDD nodes). There are two approaches to deal with this obstacle. The first approach is to modify a BDD package to make it reentrant (cf. Section 7 for details). The second approach is to utilize multiple instances of the BDD package that do not share memory. With this approach, each thread works on its own copy of related BDDs. However, changes made by one thread would not be immediately available to other threads. Hence, threads may change the BDDs (e.g., the program being synthesized) inconsistently. Therefore, we need to *merge* the results and remove/manage the inconsistencies. In this work, we consider the second approach.

Algorithm sketch. Intuitively, our algorithm works as follows. During deadlock resolution, a *master* thread spawns several *worker* threads each running on a different processor core in parallel with an instance of its own BDD package. The instance of the BDD package assigned to each worker thread is initialized using BDDs for program transitions, invariant predicate, fault-span, and fault transitions. The master thread partitions the set of deadlock states and provides each worker thread with one such partition. Subsequently, worker threads start resolving their assigned set of deadlock states in parallel by either (1) adding *safe recovery*, or (2) *eliminating* the ones (i.e., making them unreachable) from where safe recovery is not possible. Upon completion, the master thread *merges* the results returned by each worker thread and resolves inconsistencies.

5.1 Parallel Addition of Safe Recovery

Given a program P, faults F, fault-span T, invariant predicate S, safety specification $SPEC_{bt}$, and *partition predicates* $prt_1 \dots prt_n$, where $n \geq 1$ is the number of worker threads to be spawned, our goal is to synthesize a transition predicate P' such that T contains no deadlock states, i.e., $T \wedge \neg Guard(P') = false$. Before we describe our parallel algorithm for resolving deadlock states through addition of recovery actions, notice that such a recovery mechanism should not violate the safety specification. Thus, we first identify the state predicate ms (Line 2 in Algorithm ResolveDeadlockStates in Figure 1.a) from where faults alone can reach a state where $Guard(F \wedge SPEC_{bt})$ is true (i.e., faults alone can violate the safety). Now, let mt include the transitions in $SPEC_{bt}$ as well as transitions in P that end in ms. Observe that in order to ensure safety, P' (including its recovery actions) must be disjoint from mt.

After identifying the set ds of deadlock states in T (Line 4), we partition ds using the partition predicates such that $\bigvee_{i=1}^{n}(prt_i \wedge ds) = ds$. To efficiently partition deadlock states between threads, one needs to design a method such that (1) deadlock states are evenly distributed among worker threads, and (2) states considered by different threads for eliminating have a small overlap during backtracking. Regarding the first constraint, we can partition deadlock states based on values of some variable and evaluate the size of corresponding BDDs by the number of minterms that satisfy the corresponding formula. Regarding the second constraint, we expect that the overhead for such a split is as high as it requires dedicated analysis of program transitions. Hence, instead of satisfying this constraint, we add synchronization between threads. Thus, we design partition predicates based value of variables. For example, in the case of Byzantine agreement program with four worker threads, we let $prt_1 = (d.j = 0) \wedge (d.k = 0)$, $prt_2 = (d.j = 0) \wedge (d.k \neq 0)$, $prt_3 = (d.j \neq 0) \wedge (d.k = 0)$, and $prt_4 = (d.j \neq 0) \wedge (d.k \neq 0)$. Next, we assign each partition $prt_i \wedge ds$ of deadlock states to a worker thread to identify safe recovery paths from $prt_i \wedge ds$ to the invariant predicate in a layered fashion (Lines 5-8 in Algorithm ResolveDeadlockStates).

Each worker thread for adding recovery works as follows (cf. Thread AddRecovery in Figure 1.b). Let the first layer, lyr, be the invariant predicate S (Line 1). We now construct the recovery transition predicate rt by (1) including transitions that originate from the given set of deadlock states ds and end in lyr (Line 3), and (2) excluding transitions that can lead the program to a state where safety may be violated (Line 4). We add the resulting recovery transition predicate to rec (Line 5). Now, for the next iteration, we let lyr be the state predicate from where one-step safe recovery is possible (Line 6). We continue adding recovery transition predicates until no such transition predicate is added. Notice that our strategy on adding recovery paths guarantees that no cycles

Algorithm 1 ResolveDeadlockStates

Input: program P, faults F, invariant S, fault span T, safety specification $SPEC_{bt}$, and partition predicates $prt_1..prt_n$, where n is the number of worker threads.
Output: program P' and the predicate *fte* of states failed to eliminate.

1: Let *rfo* be the state predicate reachable by faults only from the invariant predicate;
2: Let *ms* be the state predicate from where faults alone can reach a state where $Guard(F \wedge SPEC_{bt})$ is true.
3: $mt := SPEC_{bt} \vee \langle ms \rangle'$;
4: $ds := T \wedge \neg Guard(P)$;

// Resolving deadlock states by adding safe recovery
5: **for** $i := 1$ **to** n **do**
6: $rt_i :=$ SpawnThread $\rightsquigarrow$ AddRecovery($ds \wedge prt_i$, S, mt);
7: **end for**
8: ThreadJoin(1..n);

9: $P := P \vee \bigvee_{i=1}^{n} rt_i$;
10: $vds, fte := false$;
11: $ds := T \wedge \neg Guard(P)$;

// Eliminating deadlock states from where safe recovery is not possible
12: **for** $i := 1$ **to** n **do**
13: $rp_i, vds_i, fte_i :=$ SpawnThread $\rightsquigarrow$ Eliminate($ds \wedge prt_i, P, S, F, T, vds, rfo, fte$);
14: **end for**
15: ThreadJoin(1..n);

// Merging results from worker threads
16: $P' := Group(\bigwedge_{i=1}^{n} rp_i)$;
17: $fte := \bigvee_{i=1}^{n} fte_i$;
18: $vds := \bigvee_{i=1}^{n} vds_i$;

19: $nds := ((T \wedge \neg S) \wedge \neg Guard(P')) \wedge \neg((T \wedge \neg S) \wedge \neg Guard(P))$;
20: $P' := P' \vee Group(P \wedge nds)$;
21: $P' := P' \vee Group(P \wedge \langle fte \wedge rfo \rangle')$;
22: **return** P', *fte*;

(a) Master Thread

Thread 1 AddRecovery

Input: deadlock states ds, invariant S, and transition predicate mt.
Output: recovery transition predicate *rec*.

1: $lyr, rec := S, false$;
2: **repeat**
3: $rt := Group(ds \wedge \langle lyr \rangle')$;
4: $rt := rt \wedge \neg Group(rt \wedge mt)$;
5: $rec := rec \vee rt$;
6: $lyr := Guard(ds \wedge rt)$
7: **until** $(lyr = false)$;
8: **return** *rec*;

Thread 2 Eliminate

Input: deadlock states ds, program P, invariant S, fault transitions F, fault span T, visited deadlock states vds, states predicate reachable by faults only *rfo*, predicate *fte* failed to eliminate.
Output: revised program transition predicate P, visited deadlock states vds, predicate *fte* failed to eliminate.

1: **wait**(*mutex*);
2: $ds := ds \ \wedge \ \neg vds$;
3: $vds := vds \ \vee \ ds$;
4: **signal** (*mutex*);
5: **if** $(ds = false)$ **then**
6: **return** P;
7: **end if**

8: $old := P$;
9: $tmp := (T \wedge \neg S) \wedge P \wedge \langle ds \rangle'$;
10: $P := P \wedge \neg Group(tmp)$;
11: $fs := Guard(T \wedge \neg S \wedge F \wedge \langle ds \rangle') \ \wedge \ \neg rfo$;
12: $P, vds, fte :=$ Eliminate($fs, P, S, F, T, vds, rfo, fte$);
13: $nds := Guard(T \wedge \neg S \wedge Group(tmp) \wedge \neg Guard(P))$;
14: $P := P \vee (Group(tmp) \wedge nds)$;
15: $nds := nds \wedge Guard(tmp)$;
16: $fte := fte \vee \neg\langle old \wedge \neg P \wedge T \wedge \langle ds \rangle' \rangle''$;
17: $P, vds, fte :=$ Eliminate($nds \wedge \neg S, P, S, F, T, vds, rfo, fte$);
18: **return** P, vds, *fte*;

(b) Worker Threads

Figure 1: Parallel algorithm for resolving deadlock states.

are introduced to the fault-span. Hence, any computation that takes a recovery path reaches the invariant predicate in a finite number of steps.

Once all worker threads complete there job (Line 8 in Figure 1.a), the master thread adds all the recovery transitions returned by worker threads to the program's transition predicate (Line 9 in Algorithm ResolveDeadlockStates). At this point, the remaining deadlock states (Line 11) have to be made unreachable, as it is not possible to add safe recovery from them to the invariant predicate.

Example (cont'd). As mentioned in the introduction, one type of deadlock states in BA is of the form $\langle \underline{0}, 0, 0, 1 \rangle$, where the Byzantine general g and non-general processes j and k agree on decision 0, but process l has decided on 1. The algorithm ResolveDeadlockStates resolves such deadlock states and their symmetrical states by adding the following recovery actions to process l (and by symmetry to processes j and k) of BA:

$$BA3_l \ :: \ d.j = 0 \ \wedge \ d.k = 0 \ \wedge d.l = 1 \ \wedge \ f.l = 0 \longrightarrow \ d.l, f.l := 0, 0|1$$
$$BA4_l \ :: \ d.j = 1 \ \wedge \ d.k = 1 \ \wedge d.l = 0 \ \wedge \ f.l = 0 \longrightarrow \ d.l, f.l := 1, 0|1$$

5.2 Parallel State Elimination

Let ds be a deadlock state predicate from where recovery to the invariant predicate cannot be added. Hence, in order for P' (the synthesized program) to satisfy the third condition of the synthesis problem, we need to ensure that ds is eliminated from the set of states that P' can reach in the presence of faults. Similar to addition of recovery paths, the Algorithm ResolveDeadlockStates launches one worker thread per each partition of ds for elimination (Lines 12-15).

The Thread Eliminate (cf. Figure 1.b) works as follows. We first keep track of *visited deadlock states* by all

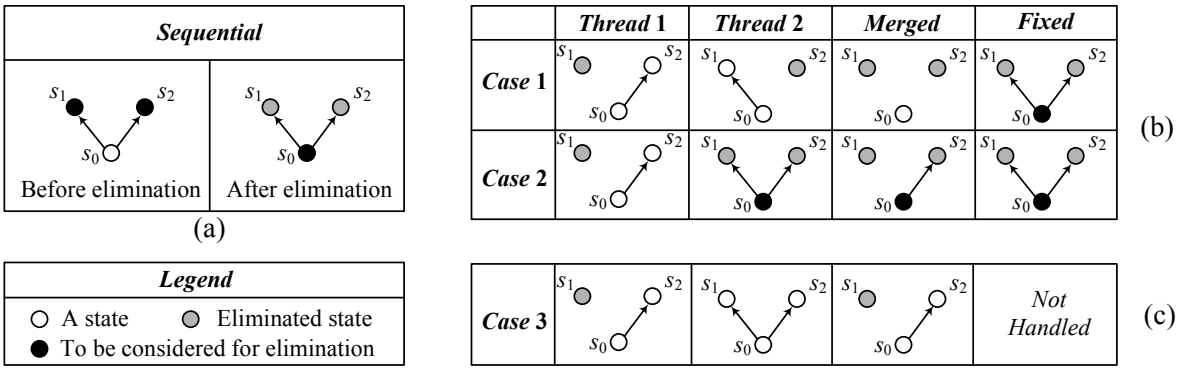

Figure 2: Inconsistencies raised by concurrency.

worker threads (Lines 1-4) so that no thread attempts to eliminate deadlock states that have already been considered for elimination. In particular, all threads synchronize on the predicate vds which contains visited deadlock states by all threads (Lines 1-4). Next, we remove all incoming transitions to ds (Lines 8-10). Then, since a program does not have control over the occurrence of faults, we eliminate states that can reach ds via a fault transition (Lines 11-12). Now, if removal of transitions in Line 10 causes some state predicate nds to become a deadlock state predicate (Line 13) then we add the transitions (and the corresponding group) that begin from nds (Lines 15-17) to P and instead, we eliminate nds[1]. We keep repeating this procedure recursively until there does not exist a state to eliminate.

Once all worker threads complete their job (Line 15 in Figure 1.a), the master thread merges all the results by collecting transitions that all worker threads agree on (Line 16). Although the above algorithm is a sound building block for a sequential algorithm, it may create inconsistencies when multiple instances of it run in parallel.

5.2.1 Handling Inconsistencies

Let s_1 and s_2 be two states that are considered for elimination and (s_0, s_1) and (s_0, s_2) be two transitions for some s_0. A sequential algorithm that applies Eliminate, removes transitions (s_0, s_1) and (s_0, s_2) which causes s_0 to be a new deadlock state (cf. Figure 2.a). Hence, it puts (s_0, s_1) and (s_0, s_2) (and corresponding group predicates) back into the program being synthesized and invokes Eliminate on state s_0. However, when multiple worker threads, say th_1 and th_2, run concurrently, there are three possible scenarios that cause inconsistencies, described next.

[1]Let P be a transition predicate. $\langle P \rangle''$ denotes the state predicate obtained by first abstracting unprimed variables in P and then replacing all primed variables of P by their corresponding unprimed variables.

Case 1. Consider the case where deadlock states s_1 and s_2 are in different partitions. Hence, th_1 invokes Eliminate on s_1 which in turn removes (s_0, s_1), and, th_2 invokes Eliminate on s_2 which removes (s_0, s_2) (cf. Figure 2.b). Thus, neither thread invokes Eliminate on s_0, since they do not identify s_0 as a deadlock state. Subsequently, when the master thread merges the results returned by th_1 and th_2 (i.e., Line 16 in Figure 1.a), s_0 becomes a new deadlock state which has to be eliminated while the group predicates of transitions (s_0, s_1) and (s_0, s_2) have been removed unnecessarily. In order to resolve this case, we replace all outgoing transitions that start from s_0 and mark s_0 as a state that has to be eliminated in subsequent iterations (Lines 19-20).

Case 2. Due to backtracking behavior of Eliminate, it is possible that th_1 and th_2 consider common states for elimination. In particular, if th_1 considers s_1 and th_2 considers both s_1 and s_2 for elimination (cf. Figure 2.b), after merging the results, no new deadlock states are introduced. However, (s_0, s_1) would be removed unnecessarily. In order to resolve this case, we collect all the states that worker threads failed to eliminate (i.e., state predicate fte in Line 17 in Figure 1.a) and replace all incoming transitions into those states (Line 21).

Case 3. It is also possible that th_1 considers s_1 and th_2 considers neither s_1 nor s_2 (cf. Figure 2.c). This case occurs when th_2 stops backtracking at a level higher than s_1 and s_2 in the reachability graph due to facing either Case 1 or Case 2. Thus, when the master thread merges the results returned by the worker threads, no new deadlock state is introduced, but (s_0, s_1) is removed unnecessarily. While identifying this case given the structures in Figure 2.c is not straightforward, one approach to resolve this inconsistency is to force all worker threads to synchronize at each backtracking step. Since such synchronization seems to decline the performance of the parallel algorithm, we choose not to handle this case. Notice that removal of (s_0, s_1) does not result in synthesizing an incorrect program. However, the program synthesized us-

ing the parallel algorithm may have less transitions than the program synthesized by the sequential algorithm. We note that this case is not due to our algorithm strategy, but an artifact of breadth-first-search nature of BDD-based reachability analysis. In fact, any random state space search strategy may as well exhibit this case.

Example (cont'd). As mentioned in the introduction, another type of deadlock states in BA is of the form $\langle \underline{0}, 0, 0, \overline{1} \rangle$, where non-general processes j and k agree with the Byzantine general on decision 0, but process l has finalized its decision on 1. Since process l has finalized its decision, we cannot resolve such deadlock states by adding safe recovery. Thus, the algorithm ResolveDeadlockStates has to eliminate states in $\langle \underline{0}, 0, 0, \overline{1} \rangle$. More specifically, the Thread Eliminate backtracks through the reachability graph until it removes the transition $\langle 1, \bot, \bot, 1 \rangle \rightarrow \langle 1, \bot, \bot, \overline{1} \rangle$. This removal creates no new deadlock state and, hence, Eliminate terminates successfully. Precisely, our algorithm revises action $BA2_l$, so that no computation of BA in the presence of faults reaches a deadlock state as follows:

$$BA2_l :: (d.l \neq \bot) \wedge (f.l = \mathit{false}) \wedge (d.j \neq \bot \vee d.k \neq \bot) \longrightarrow f.l := \mathit{true}$$

We note that in the context of of BA, inconsistency of type Case 3 does not occur. However, Cases 1 and 2 do occur, but our algorithm fixes them. In fact, the output of our synthesis algorithm is identical to the solution proposed by Lamport, Shostak, and Pease [3].

6 Experimental Results and Analysis

In this section, we present experimental results of the implementation of the Algorithm ResolveDeadlockStates. Throughout this section, all parallel experiments are run on a Sun Fire V40z with 2 dual-core Opteron processors and 16GB RAM. The BDD representation of the Boolean formulae has been done using the C++ interface to the CUDD package developed at University of Colorado [7][2]. We note that our algorithm is deterministic and the testbed is dedicated. Hence, the only nondeterministic factor in time for synthesis is synchronization among threads. Based on our experience with the synthesis, this factor has a negligible impact and, hence, multiple runs on the same data essentially reproduce the same results.

Table 1 illustrates the detailed outcome of our experiments with respect to two programs, namely, Byzantine agreement (denoted BA^i) and Byzantine agreement with fail-stop faults (denoted $BAFS^i$), where i is the number of non-general processes. In $BAFS$, in addition to Byzantine faults introduced in Section 3, the program is subject to fail-stop faults which stop normal operation of a process. Clearly, as compared to BA, $BAFS$ has a larger size of reachable states and a more complex structure. The table shows total synthesis time, state elimination time including the time spent in worker threads Eliminate and handling inconsistencies, addition of recovery time, and memory usage for synthesizing the fault-tolerant version of the given program. Recall that in addition to deadlock resolution, the total synthesis time includes other tasks such as generation of fault-span, removing unsafe actions, and reconstructing invariant which are not in the scope of this paper and, therefore, are omitted in Table 1.

6.1 Parallelism Timing Analysis

Before we analyze the results, we note that for less than 10 non-general processes, our parallel algorithm does not outperform the sequential (not threaded) algorithm due to negligible state elimination time and high level of context switching. However, for 10 or more non-general processes, as can be seen in Table 1, all results show significant speedups when our parallel algorithm runs on two or four cores as compared to the sequential algorithm. In fact, as the size of reachable states (i.e., the fault-span) grows, the parallel algorithm exhibits a better performance in both state elimination and addition of recovery. For instance, in case of $BAFS^{25}$, deadlock resolution takes more than one day using the sequential algorithm, whereas the same task can be accomplished in slightly more than 1.5 hours using the parallel algorithm running on four cores. This speedup is observed in virtually all the experiments. However, the table shows that 4-core runs do not show significant improvement over 2-core runs. We explain the reason later in this section.

One can observe that the performance improvement of our parallel algorithm is *superlinear*. Obviously, such a dramatic improvement cannot be solely attributed to parallelization. Our experiments show that this speedup is due to both *parallelization* and *partitioning* deadlock states which significantly reduces the size of BDDs involved during deadlock resolution. To understand the reason for the superlinear speedup from Table 1, we conduct three sets of experiments. First, after creating the threads, we force the threads to run sequentially by adding synchronization between them (cf. Table 2 for results). While this setup explains a part of the superlinear speedup, we find that the completion time for the case where threads run on two cores is less than half of that for the case where threads run sequentially. To understand this, we identify the size of the BDDs explored in the partitioned sequential run and in the parallel run (cf. Table 3). Furthermore, we perform a subset of experiments from Table 1 on a single processor machine

[2]Note that the results for the sequential algorithm in this paper are different from the ones appeared in [2] due to unrelated optimizations that are present in both the sequential in parallel algorithms.

	RS	*Sequential*				*Parallel 2-core*					*Parallel 4-core*				
		Tt	**El**	**Rc**	**Mm**	**Tt**	**El**		**Rc**	**Mm**	**Tt**	**El**		**Rc**	**Mm**
							Ex	Ic				Ex	Ic		
BA^{10}	10^8	0.5	0.4	0.01	18	0.2	0.1	0.02	0.02	23	0.2	0.07	0.03	0.02	36
BA^{15}	10^{12}	6.9	6.6	0.2	26	1.1	0.6	0.1	0.2	41	0.9	0.5	0.2	0.1	69
BA^{20}	10^{16}	57	55	1.2	29	5.1	3	0.4	1.4	46	4.4	2.4	0.7	0.9	75
BA^{25}	10^{20}	317	312	4	29	14.5	9.1	0.9	3.6	46	13.4	8	1.5	3	75
BA^{27}	10^{22}	538	530	5.5	32	21.4	13.5	1	5.7	46	20.4	12.3	2.1	4.5	73
BA^{28}	10^{23}	700	687	10	33	26.8	17.9	1.2	6.3	46	30.9	16	3.2	7.3	80
$BAFS^{10}$	10^9	2.9	2.7	0.1	28	0.8	0.4	0.1	0.1	25	0.8	0.3	0.1	0.1	82
$BAFS^{15}$	10^{14}	82.8	80.9	1.4	31	5.8	3	0.6	1.5	47	5.6	2.6	0.7	1.3	73
$BAFS^{20}$	10^{19}	1067	1055	9.3	34	30	18.3	2.1	7.1	54	24.9	13.9	2.3	5.2	85
$BAFS^{25}$	10^{22}	$>$ 24h	*	*	*	108.9	69.5	5.5	26.2	58	96	55	5.4	24	97
$BAFS^{27}$	10^{24}	$>$ 24h	*	*	*	147.2	94.8	6.3	35.3	58	146.7	84.1	8.03	36.2	99
$BAFS^{28}$	10^{25}	$>$ 24h	*	*	*	170.63	113.05	7.48	36.98	60	170	102	8	40.7	100

Table 1: Experimental results for algorithm ResolveDeadlockStates. **RS:** Size of reachable states. **Tt:** Total synthesis time in minutes. **El:** Total time spent in state elimination in minutes. **Ex:** Total time (m) spent by Eliminate worker threads. **Ic:** Time spent (m) for resolving inconsistencies. **Rc:** Time spent (m) for addition of recovery paths. **Mm:** Memory usage in KB.

	Sequential			*Sequential 2-partition*				*Sequential 4-partition*			
	Tt	**El**	**Rc**	**Tt**	**El**		**Rc**	**Tt**	**El**		**Rc**
					Ex	Ic			Ex	Ic	
BA^{15}	6.9	6.6	0.2	3.22	2.57	0.17	0.33	5	4.33	0.2	0.35
BA^{20}	57	55	1.2	12.58	10.22	0.38	1.62	22.87	20.62	0.48	1.55
$BAFS^{10}$	2.9	2.7	0.1	1.3	0.9	0.1	0.1	2.2	1.7	0.1	0.1
$BAFS^{15}$	82.8	80.9	1.4	16.6	13	0.8	2.1	26.4	22.6	0.8	1.9

Table 2: Effect of partitioning without parallelizing.

where no (additional) synchronization is added between the threads but they are prevented from running simultaneously because the underlying machine has only one core (cf. Table 4). These results conclusively demonstrate that the reduction in the size of BDDs caused by partitioning the deadlock states is responsible for the superlinear speedup.

In order to study the experimental results in detail consider Table 2, where we partition the set of deadlock states and then run Eliminate for each partition in a sequential manner so that the output (transition predicate) of state elimination for the first partition is input to the second invocation of Eliminate for the second partition. For instance, in case of $BAFS^{15}$, we gain $\frac{82.8}{16.6} \simeq 5$ times speedup by only splitting deadlock states in two partitions. However, Table 1 shows that the overall speedup for $BAFS^{15}$ is $\frac{82.8}{5.8} \simeq 14.3$ which means we gain $\frac{14.3}{5} \simeq 2.9$ by parallelizing on two cores. Notice that other experiments have the same pattern. There are two reasons for this extra speedup: (1) smaller size of BDDs in the parallel algorithm as compared to partitioned sequential algorithm, (2) distribution of BDDs across multiple threads. These issues are discussed next.

The effect parallelization on the size of BDDs. Table 3 shows the number of nodes in the BDD that represents *visited deadlock states* (i.e., the variable vds in Thread Eliminate in Figure 1.b) for parallel and sequential invocations of Eliminate. As can be seen, the size of nodes in the parallel runs are smaller and, hence, their manipulation is faster. This is due to the fact that when two threads are running in parallel and synchronize on vds, they do not explore the reachability graph as deep as when they are running one after another. In other words, when two Eliminates run concurrently they do not invade each other's territory. Moreover, one can observe that this behavior is more dramatic as programs get larger. As a direct result, our algorithm benefits from the synchronization on vds. We have observed this pattern

	Sequential 2-partition		*Parallel 2-core*	
	Eliminate1	Eliminate2	Eliminate1	Eliminate2
BA^{15}	4938	4943	4603	4774
BA^{20}	8943	8379	6578	6464

Table 3: Number of nodes in BDDs that represent *visited deadlock states*.

	Seq.	Par. 2-partition 1-core	Par. 4-partition 1-core
BA^{15}	6.2	1.4	2.1
BA^{20}	51.8	6	9.4

Table 4: Total synthesis time when parallel algorithm runs on a single-core machine. (Note that since this set of experiments required a single core machine, they are performed in a different setup than previous experiments. Hence, the time cannot be directly compared with time from other tables.)

in other experiments as well.

The effect of distribution of BDDs across multiple threads. As another approach to analyze the super-linear speedup, we repeated a subset of experiments presented in Table 1 on a single processor/core machine with 2.2GHz processor and 1G memory. Thus, in this setup, similar to the experiments from Table 1, deadlock states are partitioned into multiple threads. Although no explicit (additional) synchronization is added between these threads (as done in experiments in Table 2), they cannot execute simultaneously since there is only one processor/core. The results from these experiments are available in Table 4. As we can see from this table, for BA^{15} (respectively, BA^{20}), a speedup of 4.3 (respectively, 8.6) is obtained with two threads running on a single core. By comparison, in this example, the speedup was 6.3 (respectively, 11.2) when these threads were permitted to execute on a multicore machine. Thus, results from Tables 2-4 conclusively demonstrate that the super-linear speedup in Table 1 is caused by the fact that the size of the BDDs is reduced due to partitioning of deadlock states across different threads.

Table 2 also reveals why 4-core runs do not outperform 2-core runs significantly. This is due to creation of significantly more inconsistencies in a 4-partition structure than a 2-partition structure. In fact, parallelization using 4-core shows a better improvement than 2-core. Thus, our parallel algorithm is considerably efficient. Table 1 also shows that we benefited from parallelism since the time spent to resolve inconsistencies was significantly less than the time spent for running worker Eliminate threads. However, more research needs to be done on effective partitioning which is an issue in distributed model checking as well. As an example of unbalanced partitioning, we note that if one partitions deadlock states of Byzantine agreement based on $b.g$ and $d.j$, no speedup is gained, since the value of $b.g$ in all deadlock states in fault-span is 1.

We have also observed that in cases where there exist a large number of processes in a distributed program, computing group predicates becomes a bottleneck, which in turn may make the execution of worker threads into the corresponding sequential algorithm. In fact, this is the very reason that parallel addition of recovery does not show a significant performance improvement.

6.2 Memory Usage

Although incorporating multiple instances of a BDD package increases the memory usage, we argue that since the required amount of memory is not a bottleneck, the trade off between speedup and memory usage is remarkably beneficial. In fact, the crucial factor in our experiments (and perhaps in general in program synthesis) is time and not space. Moreover, Table 1 shows that instantiating two BDD packages does not double the amount of required memory.

7 Related Work

Automated program synthesis and revision has been studied from various perspectives. Inspired by the seminal work by Emerson and Clarke [8], Arora, Attie, and Emerson [9] propose an algorithm for synthesizing fault-tolerant programs from CTL specifications. Their method, however, does not address the issue of addition of fault-tolerance to existing programs. Kulkarni and Arora [10] introduce enumerative synthesis algorithms for automated addition of fault-tolerance to centralized and distributed programs. In particular, they show that the problem of adding fault-tolerance to distributed programs is NP-complete. In order to remedy the NP-hardness of synthesis of fault-tolerant distributed programs and overcome the state explosion problem, we proposed a set of symbolic heuristics [2] which allowed us to synthesize programs with state space of size 10^{30} and beyond.

Ebnenasir [11] presents a divide-and-conquer method for synthesizing *failsafe* fault-tolerant distributed programs. A failsafe program is one that does not need to satisfy its liveness specification in the presence of faults. Thus, a respective synthesis algorithm does not need to resolve deadlock states outside the invariant predicate. Moreover, Ebnenasir's synthesis method resolves deadlock states inside the invariant predicate in a sequential manner.

Parallelization of symbolic reachability analysis has been studied in the model checking community from different perspectives. In [4, 12, 13], the authors pro-

pose solutions and analyze different approaches of parallelization of *saturation*-based generation of state space in model checking. In particular, in [13], the authors show that in order to gain speedups in saturation-based parallel symbolic verification, one has to pay a penalty for memory usage up to 10 times, as compared to the sequential algorithm. Other efforts range from simple approaches that essentially implement BDDs as two-tiered hash tables [14, 15], to sophisticated approaches relying on *slicing* BDDs [16] and techniques for *workstealing* [17]. However, the resulting implementations show only limited speedups.

8 Conclusion and Future Work

In this paper, we focused on one of the main complexity barriers, *resolution of deadlock states*, in automated addition of fault-tolerance to distributed programs. Our approach was based on parallelization with multiple threads. We considered parallelization in two scenarios: (1) adding recovery transitions, and (2) eliminating deadlock states. With the parallelization of these scenarios, we gain a significant speedup. As expected, most of the speedup was due to reduction in time to eliminate deadlock states. We also demonstrated that we gained *superlinear* speedup due to partitioning deadlock states that reduces the size of corresponding BDDs.

While parallelization reduces the time spent in eliminating deadlock states, it may also lead to some inconsistencies that have to be resolved. The time for resolving such inconsistencies is one of the bottlenecks in parallelization, as this inconsistency is resolved sequentially. We note that the synchronization on *visited states* was also added, in part, to reduce inconsistencies among threads by requiring them to coordinate with each other.

Our approach provides each thread with its own copy of shared variables. Although this has a potential to increase the memory usage, our experiments show that the actual memory usage is low. In general, synthesis problems tend to have a higher time complexity than the corresponding verification problems. Hence, we expect that a symbolic synthesis algorithm will *run out of time* before it *runs out of memory*. Hence, the increased space complexity is unlikely to be the bottleneck during synthesis.

One future work in this context is to identify tradeoff in additional synchronization among threads. While this may reduce concurrency among threads, it may also reduce the time for resolving inconsistencies. Another future work is parallelization of the other complexity barrier, *fault-span* generation.

References

[1] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 130–140, 2001.

[2] B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, 2007.

[3] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[4] J. Ezekiel, G. Lüttgen, and R. Siminiceanu. Can Saturation be parallelised? on the parallelisation of a symbolic state-space generator. In *International Workshop on Parallel and Distributed Methods of Verification (PDMC)*, pages 331–346, 2006.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[6] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[7] F. Somenzi. CUDD: Colorado University Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[8] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[9] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. In *Principles of Distributed Computing (PODC)*, pages 173–182, 1998.

[10] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, 2000.

[11] A. Ebnenasir. DiConic addition of failsafe fault-tolerance. In *Automated Software Engineering (ASE)*, pages 44–53, 2007.

[12] J. Ezekiel and G. Lüttgen. Measuring and evaluating parallel state-space exploration algorithms. In *International Workshop on Parallel and Distributed Methods in Verification (PDMC)*, 2007.

[13] J. Ezekiel, G. Lüttgen, and G. Ciardo. Parallelising symbolic state-space generators. In *Computer Aided Verification (CAV)*, pages 268–280, 2007.

[14] K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel BDD package. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 501–507, 1998.

[15] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *Design automation (DAC)*, pages 641–644, 1996.

[16] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design (FMSD)*, 29(2):157–175, 2006.

[17] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 129–145, 2005.